Tram: A Token-level Retrieval-augmented Mechanism for Source Code Summarization

Anonymous ACL submission

Abstract

Automatically generating human-readable text describing the functionality of a program is the intent of source code summarization. Although neural language models achieve significant performance in this field, they are limited by their inability to access external knowledge. To address this limitation, an emerging trend is combining neural models with external knowledge through retrieval methods. Previous methods have relied on the sentence-level retrieval paradigm on the encoder side. However, this paradigm is coarse-grained, noise-filled and cannot directly take advantage of the high-quality retrieved summary tokens on the decoder side. In this paper, we propose a fine-grained Token-level retrieval-augmented mechanism (Tram) on the decoder side rather than the encoder side to enhance the performance of neural models and produce more low-frequency tokens in generating summaries. Furthermore, to overcome the challenge of token-level retrieval in capturing contextual code semantics, we also propose integrating code semantics into individual summary tokens. The results of extensive experiments and human evaluation show that our token-level retrieval-augmented approach significantly improves performance and is more interpretable.

1 Introduction

With software functions becoming more comprehensive and complex, it becomes a heavy burden for developers to understand software. It has been reported that nearly 90% (Wan et al., 2018) of effort is used for maintenance, and much of this effort is spent on understanding the maintenance task and related software source codes. Source code summary as a natural language is indispensable in software since humans can easily read and understand it, as shown in Table 1. However, manually writing source code summaries is time-consuming and tedious. Besides, the source code summary is often outdated in continuous software iteration. Hence, automatically generating concise, human-readable source code summaries is critical and meaningful.

```
def cos(x):
    np = import module("numpy")
    if isinstance(x, (int, float)):
        return interval(np.sin(x))
    elif isinstance(x, interval):
        if (not(np.isifnite(x.start) and
                np.isfinite(x.end))):
            return interval((-1), 1, is_valid=x.is_valid)
        (na, _) = divmod(x.start, (np.pi / 2.0))
        (nb, _) = divmod(x.end, (np.pi / 2.0))
        start = min(np.cos(x.start), np.cos(x.end))
        end = max(np.cos(x.start), np.cos(x.end))
        if ((nb - na) > 4):
            return interval((-1), 1, is_valid=x.is_valid)
        elif (na == nb):
            return interval(start, end, is_valid=x.is_valid)
        else:
            if ((na // 4) != (nb // 4)):
                end = 1
            if (((na - 2) // 4) != ((nb - 2) // 4)):
                start = -1
            return interval(start, end, is_valid=x.is_valid)
    else:
        raise NotImplementedError
```

Summary: evaluates the **cos** of an interval.
Token-level retrieval results
at the next generation step "cos":
cos, tangent, sin, hyperbolic, · · ·

Table 1: A sample of source code summarization.

With the development of language models and the linguistic nature of source code, researchers explored Seq2Seq architecture, such as recurrent neural networks to generate summaries (Iyer et al., 2016; Loyola et al., 2017; Liang and Zhu, 2018). Soon afterward, transformer-based models (Ahmad et al., 2020; Wu et al., 2021; Gong et al., 2022) were proposed, outperforming previous RNN-based models by a large margin. Recently, many approaches have been proposed to leverage the structural properties of source code, such as Abstract Syntax Tree (AST) and Program Dependency Graph (PDG). Current structure-aware methods typically either fuse structural information in a hybrid manner (Hu et al., 2018; Shido et al., 2019; LeClair et al., 2020; Choi et al., 2021; Shi et al., 2021), or use a structured-guided way (Wu et al., 2021; Son et al., 2022; Gong et al., 2022; Guo et al.,

2022b; Choi et al., 2023). Although these methods have shown promising results, they primarily focus on leveraging the information within the code to obtain richer code representation without fully utilizing the potential of the available human-written code-summary pairs.

In order to leverage external existing high-quality code and the corresponding summary instances, recent works (Zhang et al., 2020; Li et al., 2021; Liu et al., 2021; Parvez et al., 2021) have proposed a retrieval augmented approach. Their unified paradigm involves sentence-level retrieval, which uses text similarity metrics or code semantic similarity metrics to retrieve the most similar code snippet from a code repository for the given input code snippet. The retrieved code snippet and its corresponding summary are either directly concatenated with the input code snippet or semantically enhanced to augment the input code snippet on the encoder side. However, the granularity of sentence-level retrieval methods poses challenges. Specifically, they can erroneously retrieve and incorporate code snippets that, while syntactically similar, are semantically distinct or those that only bear partial semantic resemblance. The unintended noise introduced through such mismatches can adversely affect the generation performance, especially for low-frequency tokens. Moreover, code summarization is essentially a generative task, the decoder autoregressively generates the summary tokens. However, previous sentence-level retrieval-augmented methods neglect to fuse the retrieved information on the decoder side, only doing so on the encoder side, which will result in the utilization pattern being indirect and insufficient.

These limitations have inspired us to explore a more fine-grained and sufficient retrieval approach on the summary generation process. In order to achieve the purpose of retrieving semantic similar summary tokens on the decoder side, we first construct a datastore to store the summary tokens and corresponding representations through a pre-trained base model offline. Meanwhile, to overcome the challenge of not fully utilizing code semantics on the encoder side when retrieving on the decoder side, we intelligently fuse summary token representation with code token representation and AST node representation with attention weight. This approach fully considers contextual code semantics associated with summary tokens. Then, at each generation step, the fused summary token representation is used to retrieve the top-K most similar tokens. As illustrated in Table 1, the token-level retrieval results at the next token generation step *"cos"* are *"cos, tangent, sin, hyperbolic, $\cdots$"*. The retrieved top-K tokens are expanded to a probability distribution, which we refer to as the retrieval-based distribution. The retrieval-based distribution is then fused with the vanilla distribution to form the final distribution. Additionally, our proposed token-level retrieval mechanism can be seamlessly integrated with existing sentence-level retrieval methods and code-related large pre-trained models.

To facilitate future research, we have made our code publicly available[1]. Overall, the main contributions of this paper can be outlined as follows:

(1) We are the first to explore a Token-level retrieval-augmented mechanism (Tram) on the decoder side for source code summarization.

(2) Our proposed retrieval-augmented mechanism is orthogonal to existing improvements, such as better code representation, additional sentence-level retrieval approaches, and pre-trained models.

(3) Extensive experiments and human evaluation show that Tram significantly outperforms other baseline models, generates more low-frequency tokens and is more interpretable.

2 Related Works

Retrieval-based Source Code Summarization. Liu et al. (2021) retrieved the most similar code snippet by text similarity metric to enrich target code structure information for getting a better code representation encoder. This retrieval method only carries out from the perspective of text similarity and neglects code semantic similarity. Besides, the summary corresponding to the retrieved code snippet is just a simple concatenation to the encoder. Zhang et al. (2020); Parvez et al. (2021) used a pre-trained encoder to obtain code semantic representation, which was used to retrieve similar code snippets. The former only uses similar code snippets and discards the corresponding summaries; the latter directly splice the retrieved code snippet and the corresponding summary behind the target code; both are also aimed at better code representation on the encoder side. Different from the above sentence-level retrieval methods, Tram performs token-level retrieval augmentation at each step of the decoder that generates the next token.

[1] https://anonymous.4open.science/r/SourceCodeSummary-8ABD

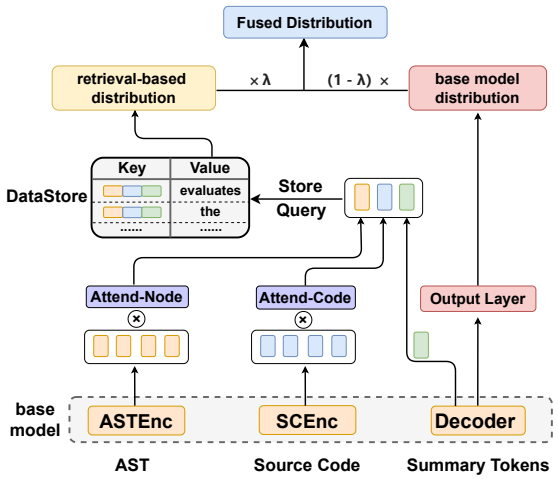

Figure 1: The overview architecture of Tram.

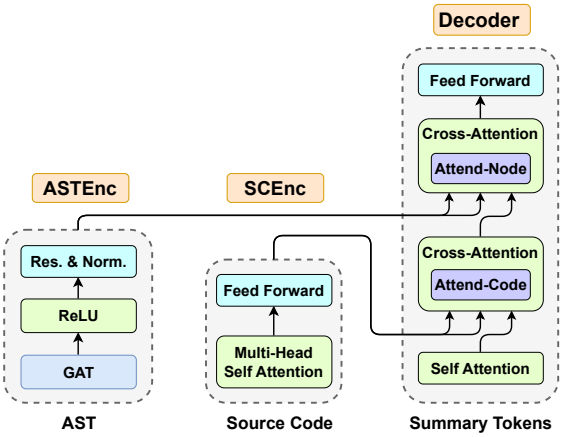

Figure 2: The architecture of base model.

K-Nearest-Neighbor Machine Translation. Recently, non-parametric methods have been successfully applied to neural machine translation (Khandelwal et al., 2021; Jiang et al., 2021; Zheng et al., 2021a,b). These approaches complement advanced NMT models with external memory to alleviate the performance degradation in domain adaption. Compared to these works, we have taken full account of the inherent structural information of the code and have intelligently integrated code semantics in the retrieval process.

3 Methodology

3.1 Overview

The overview architecture of Tram is shown in Figure 1. Initially, we introduce the base model, which is an encoder-decoder architecture that takes a code snippet and corresponding AST as input and generates a summary as output. Building upon the base model, we then construct a datastore that stores summary tokens and corresponding representations, where the representation is an intelligent combination of the decoder representation, code token representation, and AST node representation. Next, we develop a fine-grained token-level retrieval mechanism. This mechanism focuses on retrieving the top-K most similar tokens from the datastore and generating a retrieval-based distribution. The retrieval-based distribution is then fused with the vanilla base model distribution by a weight hyper-parameter λ to form the final distribution. Additionally, we detail the integration of both token-level and sentence-level retrieval. The combination of token-level retrieval and sentence-level retrieval enables a more comprehensive summarization process. In terms of integrating Tram with code pre-trained models, the implementation is broadly consistent and detailed in Appendix A.

3.2 Base Model

The base model serves as the foundation for the subsequent retrieval process. It is designed to construct the datastore and generate the base model distribution. Figure 2 illustrates the specific architecture of the base model, which consists of two encoders (SCEnc and ASTEnc) and a decoder.

Source Code Encoder (SCEnc). As shown in Figure 2, we utilize Transformer (Vaswani et al., 2017) as the encoder for the source code tokens. The Transformer consists of stacked multi-head attention and parameterized linear transformation layers. Each layer emphasizes on self-attention mechanism. Nevertheless, as pointed out in Ahmad et al. (2020), the code semantic representation is influenced by the mutual interactions between its tokens rather than their absolute positions. Therefore, we adopt the method of relative positional encoding, as proposed by Shaw et al. (2018).

Assuming the code snippet contains p tokens $[t_1, t_2, ..., t_p]$, after SCEnc, each token has a hidden representation, which is denoted as:

$$[h_1, h_2, ..., h_p] = SCEnc([t_1, t_2, ..., t_p])$$

AST Encoder (ASTEnc). Furthermore, the AST of the source code can be considered as a graph structure, making it suitable for representation and learning using Graph Neural Networks (GNNs). Taking advantage of the GAT's (Veličković et al., 2018) exceptional performance and its ability to assign adaptive attention weights to different nodes,

we employ GAT to represent each node in the AST. The graph encoder layer processes the AST by first aggregating the neighbors of the nodes with edge information. It then updates the nodes with the aggregated information from their neighborhoods.

After updating the node information, the node representations are put together into a $ReLU$ activation followed by residual connection (He et al., 2016) and layer normalization (Ba et al., 2016).

Assuming the AST of the code snippet contains q nodes $[n_1, n_2, ..., n_q]$, after the ASTEnc, each node has a hidden representation, denoted as:

$$[r_1, r_2, ..., r_q] = ASTEnc([n_1, n_2, ..., n_q])$$

Summary Decoder. The summary decoder is designed with modified transformer decoding blocks. At time step t, given the existing summary tokens $[s_1, s_2, ..., s_{t-1}]$, the decoding blocks first encode them by masked multi-head attention. After that, we expand the transformer block by leveraging two multi-head cross-attention modules to interact with the two encoders for summary decoding. One multi-head cross-attention module is performed over the code token features to get the first-stage decoded information, which will then be fed into the other over the learned AST node features for the second-stage decoding. Then the decoded summary vectors $[d_1, d_2, ..., d_{t-1}]$ are put into a feed-forward network for non-linear transformation.

3.3 Datastore Construction

Based on the base model, to achieve the goal of fine-grained token-level retrieval, we build the datastore that stores summary tokens and corresponding representations. At the stage of datastore establishment, we adopt the above pre-trained base model to go through all training instances in an offline manner. During this process, for each instance, the SCEnc and ASTEnc encode the code tokens and AST nodes into a sequence of hidden states: $[h_1, h_2, ..., h_p]$ and $[r_1, r_2, ..., r_q]$, the decoder generates the target summary autoregressively. At time step t, the decoder takes existing summary token $[s_1, s_2, ..., s_{t-1}]$ as input, for the last token s_{t-1}, the decoder's first cross-attention module gets the attention score of the code tokens (called Attend-Code $[\alpha_1, \alpha_2, ..., \alpha_p]$), the second cross-attention module gets the attention score of the AST nodes (called Attend-Node $[\beta_1, \beta_2, ...\beta_q]$). We use Attend-Code and Attend-Node to perform weighted summation of the representations of code tokens and AST nodes, respectively, denoted as:

$$[\alpha_1, \alpha_2, ..., \alpha_p] * [h_1, h_2, ..., h_p]^T = H_t$$

$$[\beta_1, \beta_2, ..., \beta_q] * [r_1, r_2, ..., r_p]^T = R_t$$

where H_t means weighted code token representation, R_t means weighted AST node representation.

After two cross-attention modules, the input token s_{t-1} is converted to token representation d_{t-1}. Because the goal at time step t is to generate the next token s_t, we pick the token representation d_{t-1} to represent s_t. To fully consider the contextual code semantics associated with the summary token, we concatenate H_t, R_t, and d_{t-1} to create the final and more comprehensive representation of s_t. Besides, to facilitate efficient retrieval in the subsequent steps, we applied L_2 regularization to the representations in practice, denoted as:

$$k_t = Concat(H_t, R_t, d_{t-1})$$

$$\widetilde{k}_t = L_2_Normalize(k_t)$$

where $\widetilde{k}_t$ is the final presentation of token s_t. Finally, the ground-truth summary token s_t and corresponding representation $\widetilde{k}_t$ are inserted into datastore as a key-value pair, denoted as (key, value) = ($\widetilde{k}_t$, s_t), the whole datastore can be denoted as:

$$(\mathcal{K}, \mathcal{V}) = \{(\widetilde{k}_t, s_t), \forall s_t \in S\}$$

where S means all summary tokens in the training dataset. It is important to note that the datastore contains duplicate tokens because the same summary token can have different keys, representing different semantic representations due to variations in linguistic contexts.

3.4 Token-level Retrieval

During inference, at each decoding step t, the current summary token representation d_{t-1} is combined with the corresponding H_t and R_t using the same concatenate and L_2 regularization operator as query q_t. The query retrieves the top-K most similar summary tokens in the datastore according to cosine similarity distance. It is worth noting that we use cosine similarity instead of squared-L^2 distance because of the performance of the preliminary experiment. As an added bonus, cosine similarity can be seen as retrieval confidence. In practice, the retrieval over millions of key-value pairs is carried out using FAISS (Johnson et al., 2019), a library for fast nearest neighbor search in high-dimensional spaces. The retrieved key-value pairs (k, v) and corresponding

cosine similarity distance α composed a triple set $\mathcal{N} = \{(k_i, v_i, \alpha_i) | i = 1, 2, \cdots, K\}$. Inspired by KNN-MT (Khandelwal et al., 2021), the triple set can then be expanded and normalized to the retrieval-based distribution as follows:

$$P_r(s_t|c, \hat{s}_{<t}) \propto \sum_{(k_i, v_i, \alpha_i) \in \mathcal{N}} \mathbb{1}_{v_i = s_t} \exp\left(g(k_i, \alpha_i)\right)$$

$$g(k_i, \alpha_i) = \alpha_i * T$$

where $g(\cdot)$ can be any Kernel Density Estimation (KDE); in practice, we use the product form; T is the temperature to regulate probability distribution.

3.5 Fused Distribution

The final prediction distribution can be seen as a combination of the vanilla base model output distribution and the retrieval-based distribution, which is interpolated by a hyper-parameter λ:

$$\begin{aligned} P(s_t|c, \hat{s}_{<t}) = \lambda * P_r(s_t|c, \hat{s}_{<t}) \\ + (1 - \lambda) * P_m(s_t|c, \hat{s}_{<t}) \end{aligned}$$

where P_m indicates the base model distribution.

3.6 Additional Sentence-level Retrieval

Our proposed token-level retrieval augmented method can also be seamlessly incorporated with additional sentence-level retrieval. Sentence-level retrieval here means using the target code snippet to retrieve the most semantically similar code snippet in the corpus through code semantic representations. Then we assign an additional but the same base model for the most similar code snippet to generate tokens autoregressively. At each generation step, the decoder of the additional base model (generating similar-code-based next token distribution) is synchronous with the original target code snippet decoder (generating base model next token distribution). Finally, the above two distributions, together with the "token-level retrieved next token distribution", form the final distribution through a weighted sum, which is denoted as:

$$\begin{aligned} P(s_t|c, \hat{s}_{<t}) = \lambda_1 * P_r(s_t|c, \hat{s}_{<t}) \\ + \lambda_2 * Sim * P_s(s_t|\langle c \rangle, \hat{s}_{<t}) \\ + (1 - \lambda_1 - \lambda_2) * P_m(s_t|c, \hat{s}_{<t}) \end{aligned}$$

where P_s is the additional base model produced distribution, $\langle c \rangle$ is the most semantically similar code snippet to the target code snippet c, and Sim is the corresponding similarity score.

Datasets	Java	Python	CCSD	Python‡
Train	69,708	55,538	84,316	65,236
Validation	8,714	18,505	4,432	21,745
Test	8,714	18,502	4,203	21,745
Code: Avg. tokens	73.76	49.42	68.59	150.82
Summary: Avg. tokens	17.73	9.48	8.45	9.93

Table 2: Statistics of the experimental datasets.

4 Experiments

4.1 Experimental Setup

Datasets. We conduct the experiments on four public benchmarks of Java (Hu et al., 2018), Python (Wan et al., 2018), CCSD (C Code Summarization Dataset) (Liu et al., 2021), and Python‡ (Zhang et al., 2020). The partitioning of train/validation/test sets follows the original datasets. The statistics of the four datasets are shown in Table 2.

Out-of-Vocabulary. The vast operators and identifiers in program language may produce a much larger vocabulary than natural language, which can cause Out-of-Vocabulary problem. To avoid this problem, we apply $CamelCase$ and $snake_case$ tokenizers that are consistent with recent works (Gong et al., 2022; Wu et al., 2021; Ahmad et al., 2020) to reduce the vocabulary size of source code.

Metrics. Similar to recent work (Gong et al., 2022; Son et al., 2022), we evaluate the source code summarization performance using three widely-used metrics, BLEU (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005) and ROUGE-L (Lin, 2004). Furthermore, considering the essence of source code summarization to help humans better understand code, we also conduct a **human evaluation** study. The volunteers are asked to rank summaries generated from the anonymized approaches from 1 to 5 (i.e., 1: Poor, 2: Marginal, 3: Acceptable, 4: Good, 5: Excellent) based on ***Similarity***, ***Relevance***, and ***Fluency*** metrics. Further details on human evaluation can be found in Appendix C.

Training Details. We implement our approach based on JoeyNMT (Kreutzer et al., 2019). The batch size is set to 32 and Adam optimizer is used with an initial learning rate 10^{-4}. For Faiss (Johnson et al., 2019) Index, we employ IndexFlatIP and top-K=16 to maintain a balance between retrieval quality and retrieval speed in the large-scale datastore. It is worth noting that only the base model requires training, and once trained, all the parameters of the base model are fixed.

Model	Java			Python		
	BLEU	ROUGE-L	METEOR	BLEU	ROUGE-L	METEOR
Transformer-based Methods						
Transformer (Ahmad et al., 2020)	44.58	54.76	26.43	32.52	46.73	19.77
CAST (Shi et al., 2021)	45.19	55.08	27.88	-	-	-
mAST + GCN (Choi et al., 2021)	45.49	54.82	27.17	32.82	46.81	20.12
SiT (Wu et al., 2021)	45.70	55.54	27.55	33.46	47.50	20.28
SiT + PDG (Son et al., 2022)	46.86	56.69	-	-	-	-
CODESCRIBE (Guo et al., 2022b)	46.93	56.18	29.13	34.44	49.02	20.91
Our Method						
Base	46.84	56.92	28.71	34.20	48.37	20.99
Tram w/o HR	47.85	57.51	29.28	35.37	49.31	21.53
Tram	**48.32** (1.48↑)	**58.13**	**29.56**	**35.97** (1.77↑)	**49.92**	**22.09**
Tram with SenRe	*48.58*	*58.43*	*29.77*	*36.23*	*50.04*	*22.23*
Our Method on Pre-trained Models						
CodeT5 (Wang et al., 2021)	46.47	58.11	27.92	35.37	51.27	23.22
CodeT5 + Tram	47.85 (1.38↑)	59.32	28.75	36.23 (0.86↑)	52.08	24.13
UniXcoder (Guo et al., 2022a)	45.32	56.61	26.52	35.89	51.17	23.11
UniXcoder + Tram	46.17 (0.85↑)	57.22	26.94	36.45 (0.56↑)	51.78	23.55

Table 3: Comparison of the performance of our method with other baseline methods on Java and Python benchmarks in terms of BLEU, ROUGE-L, and METEOR. The results of baseline models are reported in their original papers. '-' refers to no corresponding value from the paper. HR refers to code token and AST node representation; SenRe refers to additional sentence-level retrieval. All of our results are the mean of 5 runs with different random seeds.

4.2 Baselines

Transformer-based. Transformer (Ahmad et al., 2020) is the first attempt to use transformer architecture in this field. Soon, structure-aware methods were proposed. Among these are CAST (Shi et al., 2021) and mAST+GCN (Choi et al., 2021), which integrate structural information in a hybrid manner. SiT (Wu et al., 2021), SiT+PDG (Son et al., 2022), and CODESCRIBE (Guo et al., 2022b) utilize a structured-guided way. The detailed description of these baselines is shown in Appendix B.

Retrieval-based. Rencos (Zhang et al., 2020) is the first retrieval-based Seq2Seq model, which computes a joint probability conditioned on both the original source code and the retrieved most similar source code for a summary generation. HGNN (Liu et al., 2021) is the retrieval-based GNN model, which retrieval the most similar code and uses a Hybrid GNN by fusing static graph and dynamic graph to capture global code graph information.

4.3 Main Results

The main experiment results are shown in Table 3 and Table 4 in terms of three automatic evaluation metrics. The reason we have two tables is that transformer-based works compare their performance on the widely-used Java and Python benchmarks, while the retrieval-based works use two different benchmarks, namely CCSD and Python[‡]. Thus, our experiments are performed on all four datasets for a more thorough comparison. We calculate the metric values following the same scripts[2].

From Table 3, SiT + PDG and CODESCRIBE achieve better results than all previous works. However, it is worth noting that even our base model can achieve comparable performance to other models. This is due to the improved training method we used, Pre-LN (layer normalization inside the residual blocks), which is discussed in (Liu et al., 2020). This method enhances the stability of the training process and leads to better performance. Tram further boosts results with 1.39 BLEU points on Java and 1.53 BLEU points on Python and achieves new state-of-the-art results. We also observe that the performance improvement for Python is better than that for Java. The main reason we speculate is that Java has a longer average code token length (from Table 2) and richer code structure information.

In Table 4, we compare Tram with other retrieval-based models on CCSD and Python[‡] benchmarks. Our base model is even superior to other retrieval-based methods; the main reason is that the backbone [3] are different. We reproduce Rencos architecture[4] in our base model for a fair comparison, which we denoted as "Base + Rencos". Tram out-

[2] https://github.com/gingasan/sit3/blob/main/c2nl/eval/bleu/google_bleu.py

[3] Other retrieval-based methods are RNN-based.

[4] HGNN code is not open source.

Model	CCSD			Python‡		
	BLEU	ROUGE-L	METEOR	BLEU	ROUGE-L	METEOR
Retrieval-based Methods						
Rencos (Zhang et al., 2020)	14.80	31.41	14.64	34.73	47.53	21.06
HGNN (Liu et al., 2021)	16.72	34.29	16.25	-	-	-
Our Method						
Base	17.82	35.33	16.71	34.85	48.84	21.49
Base + Rencos	19.43	36.92	17.69	35.26	49.25	22.07
Tram w/o HR	21.27	37.61	18.09	36.41	50.18	22.24
Tram	**21.48** (3.66↑)	**37.88**	**18.35**	**36.73** (1.88↑)	**50.35**	**22.53**
Tram with SenRe	*22.23*	*38.16*	*18.96*	*36.95*	*50.69*	*22.93*

Table 4: Comparison of other retrieval methods. HR means code token and AST node representation; SenRe means additional sentence-level retrieval. All of our results are the mean of 5 runs with different random seeds.

Model	Java			Python‡		
	Similarity	Relevance	Fluency	Similarity	Relevance	Fluency
Rencos	-	-	-	3.07	3.06	3.96
CODESCRIBE	3.67	3.72	4.16	-	-	-
Base	3.62	3.64	4.10	3.20	3.24	4.03
Tram	**3.83**	**3.89**	**4.23**	**3.33**	**3.44**	**4.14**

Table 5: Human Evaluation on Java and Python‡ datasets.

performs all other retrieval-based methods, further improving performance with 2.05 BLEU points and 1.47 BLEU points on CCSD and Python‡, respectively. Furthermore, as shown in Table 3 and 4, enhancing Tram with additional sentence-level retrieval (refer as "Tram with SenRe") and its integration with code pre-trained models (*"Our Method on Pre-trained Models"* section in Table 3) leads to a notable improvement in performance.

4.4 Ablation Study

To validate the effectiveness of intelligently fusing summary token representation with code token representation H_t and AST node representation R_t, we conduct an ablation experiment where we eliminate the H_t, R_t, and directly use d_{t-1} to represent target summary token s_t for comparison (refer as "Tram w/o HR"). As shown in Table 3 and 4, the performance declined by 0.47, 0.60, 0.21, and 0.32 BLEU points for Java, Python, CCSD, and Python‡, respectively. This decline in performance across all datasets demonstrated the importance of fusing code semantics into the summary token for effective token-level retrieval on the decoder side.

4.5 Human Evaluation

We perform a human evaluation (details provided in Appendix C) to assess the quality of the generated summaries by Tram, Rencos, CODESCRIBE, and base model in terms of *Similarity*, *Relevance*, and *Fluency* as shown in Table 5. The results show that

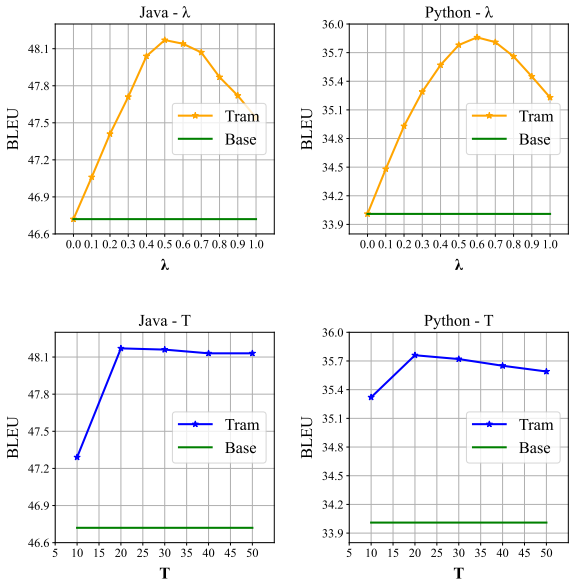

Figure 3: λ and T selections in Java and Python datasets.

Tram can generate better summaries that are more similar to the ground truth, more relevant to the source code, and more fluent in naturalness.

5 Analysis

5.1 Hyperparameters Analysis

Tram has two primary hyperparameters: λ and T. λ means the weight of the retrieval-based distribution component in the final distribution; the higher value indicates greater reliance on retrieval results, and

```
void scsi_netlink_init(void){
    struct netlink_kernle_cfg cfg;
    cfg.input = scsi_nl_rcv_msg;
    cfg.groups = SCSI_NL_GPRP_CNT;
    scsi_nl_sock = netlink_kernel_create(&init_net,
    NETLINK_SCSITRANSPORT, &cfg);
    if (!scsi_nl_sock){
        printk(KERN_ERR "%s: register of receive handler failed\n", __func__);
        return;}
    return;}
```

Base: called by scsi netlink initialization to register the scsi netlink interface.
Rencos: called by scsi netlink interface to register the scsi netlink interface.
Tram: called by scsi **subsystem** to register the scsi transport netlink interface.
Human Written: called by scsi **subsystem** to initialize the scsi transport netlink interface.
Retrieval Results: "*subsystem*" (0.90), "*transport*"(0.04), "*stack*"(0.02), "*command*"(0.0034), "*device*"(0.0025) · · ·

Table 6: A Python instance. The bold red font is the keyword of generated summary. The ***Retrieval Results*** line is the visible retrieval results and corresponding probability after applying $softmax$ on the keyword generation step.

Token Frequency		1	2	5	10	50	100
Java	Base	126	75	45	27	28	16
	Rencos	243	138	73	38	37	18
	Tram	307	164	115	51	42	21
Python‡	Base	452	376	272	176	84	82
	Rencos	799	515	344	223	88	109
	Tram	983	647	405	298	103	121

Table 7: Number of correctly generated low-frequency tokens

vice versa. T means temperature, which smooths the retrieval-based distribution. We plot the performance of Tram with different hyperparameter selections in Figure 3. The value of λ has a significant impact on the final performance, and we find that different datasets have different optimal values (i.e., $\lambda = 0.5$ for Java and $\lambda = 0.6$ for Python). We also observe that $\lambda = 1$ outperforms $\lambda = 0$. The reason is related to the BLEU score (detailed cause analysis provided in Appendix D). Regarding T, if it is too small, the retrieval-based distribution cannot be adequately distinguished; while if it is too large, the retrieval-based distribution will concentrate on a single token. Our final results indicate that both extremes result in a performance decrease.

5.2 Token Frequency In-Depth Analysis

Compared to the coarse-grained retrieval approach at the sentence-level, the token-level retrieval can capture the top-K most semantically relevant tokens at every step. This can increase the likelihood of generating those low-frequency tokens in the summary text. Since these low-frequency tokens and their corresponding representations are stored in the datastore, by retrieving the most semantically similar tokens at each generation step, these low-frequency tokens can be more easily and directly fetched from the datastore compared to purely model generated. We further conduct an in-depth statistical analysis of the generation quantity of low-frequency tokens. We first collect all the correctly generated tokens according to the ground-truth summaries. Then we count the frequencies of all these correct tokens in the training set and record the number of the correct and low-frequency tokens (frequency = 1, 2, 5, 10, 50, 100). From Table 7, we can see that Tram can correctly predict more low-frequency tokens than Rencos (sentence-level retrieval) and Base (vanilla model generated) when the token frequency is small (≤ 100).

5.3 Qualitative Analysis

We provide a python example in Table 6 to demonstrate the effectiveness and interpretability of Tram. The qualitative analysis reveals that, compared to other models, Tram enables visualization of the *Retrieval Results* and corresponding probability at each generation step, as depicted in the last line, making our approach more interpretable. More visualized instances can be found in Appendix E.

6 Conclusion

In this paper, we propose a novel token-level retrieval-augmented mechanism for source code summarization. By a well-designed fine-grained retrieval pattern, Tram can effectively incorporate external human-written code-summary pairs on the decoder side. Extensive experiments and human evaluation show that Tram not only significantly improves performance but also generates more low-frequency tokens and enhances interpretability.

Limitations

Our retrieval-augmented method (Tram) takes full advantage of external retrieval information, and the performance improvement relies on high-quality code-summary token-level pairs. However, there exists some noise in the datastore which will bias the final token distribution; therefore, dealing with noise deserves our deeper exploration. Furthermore, our experiments are only on high-resource programming language (Python, Java, C) scenarios; exploring how to apply our model in a low-resource programming language (Ruby, Go, etc.) is our future direction.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

YunSeok Choi, JinYeong Bak, CheolWon Na, and Jee-Hyong Lee. 2021. Learning sequential and structural information for source code summarization. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 2842–2851, Online. Association for Computational Linguistics.

YunSeok Choi, Hyojun Kim, and Jee-Hyong Lee. 2023. BLOCSUM: Block scope-based source code summarization via shared block representation. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 11427–11441, Toronto, Canada. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Zi Gong, Cuiyun Gao, Yasheng Wang, Wenchao Gu, Yun Peng, and Zenglin Xu. 2022. Source code summarization with structural relative position guided transformer. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 13–24.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022a. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Juncai Guo, Jin Liu, Yao Wan, Li Li, and Pingyi Zhou. 2022b. Modeling hierarchical syntax structure with triplet position for source code summarization. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 486–500, Dublin, Ireland. Association for Computational Linguistics.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 200–210, New York, NY, USA. Association for Computing Machinery.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics.

Qingnan Jiang, Mingxuan Wang, Jun Cao, Shanbo Cheng, Shujian Huang, and Lei Li. 2021. Learning kernel-smoothed machine translation with retrieved examples. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7280–7290, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547.

Urvashi Khandelwal, Angela Fan, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2021. Nearest neighbor machine translation. In *International Conference on Learning Representations*.

Julia Kreutzer, Jasmijn Bastings, and Stefan Riezler. 2019. Joey NMT: A minimalist NMT toolkit for novices. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP): System Demonstrations*, pages 109–114, Hong Kong, China. Association for Computational Linguistics.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 184–195, New York, NY, USA. Association for Computing Machinery.

Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. Editsum: A retrieve-and-edit framework for source code summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 155–166.

Yuding Liang and Kenny Zhu. 2018. Automatic generation of text descriptive comments for code blocks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1).

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. 2020. Understanding the difficulty of training transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5747–5763, Online. Association for Computational Linguistics.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-augmented generation for code summarization via hybrid {gnn}. In *International Conference on Learning Representations*.

Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 287–292, Vancouver, Canada. Association for Computational Linguistics.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA. Association for Computational Linguistics.

Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Ehud Reiter. 2018. A structured review of the validity of BLEU. *Computational Linguistics*, 44(3):393–401.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1073–1083, Vancouver, Canada. Association for Computational Linguistics.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana. Association for Computational Linguistics.

Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. CAST: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 4053–4062, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8.

Jikyoeng Son, Joonghyuk Hahn, HyeonTae Seo, and Yo-Sub Han. 2022. Boosting code summarization by embedding code structures. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 5966–5977, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In *International Conference on Learning Representations*.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on*

Automated Software Engineering, ASE 2018, page 397–407, New York, NY, USA. Association for Computing Machinery.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code summarization with structure-induced transformer. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 1078–1090, Online. Association for Computational Linguistics.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1385–1397, New York, NY, USA. Association for Computing Machinery.

Xin Zheng, Zhirui Zhang, Junliang Guo, Shujian Huang, Boxing Chen, Weihua Luo, and Jiajun Chen. 2021a. Adaptive nearest neighbor machine translation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 368–374, Online. Association for Computational Linguistics.

Xin Zheng, Zhirui Zhang, Shujian Huang, Boxing Chen, Jun Xie, Weihua Luo, and Jiajun Chen. 2021b. Non-parametric unsupervised domain adaptation for neural machine translation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 4234–4241, Punta Cana, Dominican Republic. Association for Computational Linguistics.

A Integration of Tram with Code Pre-trained Models

We need to clarify that our Tram can be integrated with generative code pre-trained models (encoder-decoder architecture), such as CodeT5 (Wang et al., 2021) and UniXcoder (Guo et al., 2022a), but is not suitable for code pre-trained models used for code understanding (encoder-only architecture), like CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2021).

Specifically, the integration process is similar to the Methodology section and primarily consists of three steps:

(1) We use Java (Hu et al., 2018) and Python (Wan et al., 2018) datasets to fine-tune the code pre-trained models, respectively, and treat the fine-tuned models as base models;

(2) During the datastore establishment phase, the process aligns with that described in the Datastore Construction section. However, we have omitted the AST input to satisfy the input conditions of the code pre-trained models;

(3) Token-level Retrieval: The retrieved top-K tokens are expanded to a probability distribution (which we refer to as the retrieval-based distribution). Then we fused the retrieval-based distribution with the vanilla distribution built on the original vocabulary table of the code pre-trained models to obtain the final distribution.

B Details on Transformer-based Methods

Transformer (Ahmad et al., 2020) is the first attempt to use transformer architecture, equipped with relative positional encoding and copy mechanism (See et al., 2017), effectively capturing long-range dependencies of source code. CAST (Shi et al., 2021) hierarchically splits a large AST into a set of subtrees and utilizes a recursive neural network to encode the subtrees. The aim is to capture the rich information in ASTs. mAST + GCN (Choi et al., 2021) adopt the AST and graph convolution to model the structural information and the transformer to model the sequential information. SiT (Wu et al., 2021) incorporates a multi-view graph matrix into the transformer's self-attention mechanism. SiT + PDG (Son et al., 2022) points program dependency graph is more effective for expressing the structural information than AST. CODESCRIBE (Guo et al., 2022b) model the hierarchical syntax structure of code by introducing a novel triplet position.

C Human Evaluation

In our human evaluation, we invited 3 PhD students and 5 master students with at least 2-5 years of software engineering experience as volunteers. We conduct a small-scale random dataset (i.e., 100 random Java samples and 100 random Python samples). The volunteers are asked to rank summaries generated from the anonymized approaches from 1 to 5 (i.e., 1: Poor, 2: Marginal, 3: Acceptable, 4: Good, 5: Excellent) based on the three following questions:

- **Similarity**: How similar of generated summary and ground truth?
- **Relevance**: Is the generated summary relevant to the source code?

- **Fluency**: Is the generated summary syntactically correct and fluent?

For each evaluation summary, the rating scale is from 1 to 5, where a higher score means better quality. Responses from all volunteers are collected and averaged.

D Cause Analysis: Performance Superiority of $\lambda = 1$ over $\lambda = 0$

λ means the weight of the retrieval-based distribution component in the final distribution. The reason is related to the BLEU score. The BLEU metric measures the similarity between two sentences by assessing the overlap of words between them. Model-generated sentences tend to produce more common words, leading to better fluency; in contrast, sentences generated through retrieval methods are more likely to include factual terms, which, when evaluated using the BLEU score, results in a higher score (Reiter, 2018). However, it may scarify the language quality.

For example, given the ground truth **"start a source file within a compilation unit."**, the retrieval-based generation with $\lambda = 1$: **"start file within a compilation unit unit."**, achieves a BLEU score of 48.78. This is higher than the model-based generation with $\lambda = 0$: **"start the source file within the unit."**, which scores a BLEU of 33.17. Indeed, neither $\lambda = 1$ or $\lambda = 0$ is good enough, and we need a trade-off between the retrieval and the model generation.

E Qualitative Examples

Table 8 shows a couple of qualitative examples to demonstrate the effectiveness and interpretability of Tram.

```c
void batadv_sysfs_del_meshif(struct net_device *dev)
{
    struct batadv_priv *bat_priv = netdev_priv(dev);
    struct batadv_attribute **bat_attr;
    for (bat_attr = batadv_mesh_attrs; *bat_attr; ++bat_attr)
        sysfs_remove_file(bat_priv->mesh_obj, &((*bat_attr)->attr));

    kobject_uevent(bat_priv->mesh_obj, KOBJ_REMOVE);
    kobject_del(bat_priv->mesh_obj);
    kobject_put(bat_priv->mesh_obj);
    bat_priv->mesh_ojb = NULL;
}
```

Base: Remove mesh interface-related sysfs sysfs entries.
Rencos: Delete mesh junction sysfc attributes.
Tram: Remove soft **interface** specific sysfs entries.
Human Written: Remove soft **interface** specific sysfs entries.
Retrieval Results: "*interface*" (0.82), "*portal*"(0.11), *"bridge"*(0.04), *"junction"*(0.0086), *"link"*(0.0013) · · ·

```python
def category_structure(category, site):
    return {'description': category.title,
    'html_Url': ('%s://%s%s'%(PROTOCOL, site.domain,
                category.get_absolute_url())),
    'rss_Url': ('%s://%s%s'%(PROTOCOL, site.domain,
                reverse('zinnia:category_feed', args=[category.tree_path]))),
    'category_Id': category.pk ,
    'parent_Id': ((category.parent and category.parent.pk) or 0 ),
    'category_Description': category.description,
    'category_Name': category.title }
```

Base: updates the structure.
Rencos: a post structure.
Tram: a **category** structure.
Human Written: a **category** structure.
Retrieval Results: "*category*"(0.43), "*tag*"(0.11), "*post*"(0.07), "*helper*"(0.06), "*version*"(0.06) · · ·

Table 8: Task samples. The first is a C instance; the second is a Python instance. The bold red font is the keyword of the generated summary. The ***Retrieval Results*** line is the visible retrieval results and corresponding probability after applying $softmax$ on the keyword generation step.